1-1-2003

Hoard: A Peer-to-Peer Enhancement for the Network File System

Ali Raza Butt

Troy A. Johnson

Yili Zheng

Y. Charlie Hu

Hoard: A Peer-to-Peer Enhancement for the Network File System*

Ali Raza Butt

Troy A. Johnson

Yili Zheng

Y. Charlie Hu

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

*This research is supported in part by an NSF CAREER award (ACI-0238379).

Contents

Abstract

This work presents Hoard, a peer-to-peer enhancement for the widely-used Network File System (NFS). Certain features–such as location transparency, mobility transparency, load balancing, and file replication–are missing from NFS but are commonly found in peer-to-peer storage applications. On the other hand, features that NFS provides–specifically hierarchical file organization, directory listings, and file permissions–are missing from most peer-to-peer storage applications. Hoard strives to provide a low-overhead enhancement for NFS through a NFS server that leverages a peer-to-peer routing substrate for efficient node communication. By blending the strengths of NFS with those of peer-to-peer, Hoard provides a single, reliable, shared file system that acts as a large redundant storage with normal NFS semantics.

We present the design, implementation, and evaluation of the Hoard file system. Our experiments show that, for an eight-node system, Hoard adds 25% overhead to unmodified NFS, it achieves load balancing in distributed directories, and it guarantees 99.99% *or better file availability using the availability trace of machines in a large organization over a period of 35 days.*

1 Introduction

This work presents Hoard, a peer-to-peer enhancement for the widely-used Network File System (NFS) [30, 7]. Hoard provides a single file system image identical to a NFS file system, yet offers features commonly found in peer-to-peer storage systems [28, 13], such as location transparency, mobility transparency, load balancing, and high availability through file replication and transparent fault handling. Hoard leverages peer-to-peer technology and the unused disk space of desktop machines to enhance NFS. It does not entail changes to the underlying operating system, and requires only minimal configuration. The result is a simple yet effective system, which is readily deployed, does not burden the user with the need to learn a new interface, and supports unmodified applications.

The design of Hoard is aimed at academic and corporate networks on the order of 10^4 nodes, where NFS and its cross-mounting facilities are extensively used to provide users access to storage beyond their local disk. In such environments, efficiency and economics dictate the widespread use of off-the-shelf desktops for fulfilling the computing needs. These machines have become increasingly powerful, both in terms of processing power and storage capacity [17], and therefore, have the potential to be used as shared resources if so desired. Unfortunately, a large amount of free disk space that exists on typical desktops is wasted as individual users are mostly served by central NFS servers, which are easier to maintain and can also provide failure resiliency via backups.

One way to harness this unused disk space is to run NFS servers on all the machines with redundant disk space, and let the users share the space. However, simply running NFS servers on all the nodes in a network is not the solution; the maintenance of a huge number of servers can be inhibiting, and human interaction and configuration errors may poorly affect the performance. Furthermore, if all nodes are NFS servers, users must remember which machines their files are on, and that may be difficult if more than a few machines are accessed. Symbolic links can help the user to locate their files quickly, but stale links can make the situation even more confusing. Another issue is that NFS does not provide redundancy, so if machines fail or are taken offline for maintenance, the information stored on them becomes inaccessible. The failure often causes other machines (repeatedly trying to access the failed machines) to respond slowly to requests they receive – an effect which spreads rapidly to degrade the performance of the entire system. The user may retrieve files from a daily or weekly backup storage, but in a large organization it may not be economically feasible to backup the data from the local disks of all

machines.

It seems desirable to solve these problems, though likely it is not practical to replace NFS in an established computing environment. (We also dismiss switching to AFS [18] or xFS [4] for this reason.) The NFS `automounter` [8] can help solve some of the issues of maintaining a large number of NFS servers. It can help to mount file systems on demand in order to eliminate the need to keep all of the file systems permanently mounted. However, the `automounter` cannot leverage the redundancy provided by the distributed disk space, nor is it capable of maintaining replicas on its own.

Hoard addresses these issues, and provides additional features of fault tolerance and high availability, which come naturally from the use of peer-to-peer systems. Since the widespread use of NFS is indispensable in the targeted environments, the key idea here is to extend NFS without incurring any changes to the underlying file system. Specifically, we organize the nodes that contribute disk space into a peer-to-peer overlay which then uses NFS to replicate files across peers and make the location of the files transparent to the user. The peer-to-peer system is built on top of Pastry [27], which provides a mechanism for routing and object location in a self-administered peer-to-peer overlay network. Hoard uses Pastry to transparently locate the nodes where individual files are stored, employs NFS mechanisms to provide access to files, and combines features of both to maintain replicas for fault tolerance. By blending the strengths of NFS with those of peer-to-peer overlay networks, Hoard provides a large redundant storage with standard NFS semantics. Hoard provides an effective solution that is easy to install and use, without the need for any changes to the existing file systems or extensive administrative involvement.

The main contributions of this work are as follows:

1. the aggregation of unused disk space on many computers into a single, shared file system,
2. location transparency,
3. mobility transparency (i.e., transparent migration of files and subdirectories),
4. load balancing,
5. high availability through replication and transparent fault handling, and

6. a detailed evaluation of the approach, including its performance compared to unmodified NFS, and its ability to provide load balancing and fault tolerance.

The rest of the report is organized as follows. Section 2 discusses the motivation and the enabling technologies that make our proposed approach possible. Section 3 presents the detailed design of Hoard and how various NFS operations are supported. Section 4 presents an evaluation of the system. Section 5 discusses related work. Finally, Section 6 presents concluding remarks.

2 Enabling technologies

There are two aspects of advancement in hardware technology and peer-to-peer routing algorithms that serve as enabling technologies for our proposed approach: availability of a large amount of free disk space on desktops and compute servers, and existence of efficient peer-to-peer routing algorithms. In the following sections we discuss these aspects in more detail.

2.1 Large unused local disk space on desktops

Most desktop computers in today's academic or corporate environments are purchased mainly for processing power. However, standard packages, which are rampant in such environments, usually ship with large-capacity disk drives [14, 12]. In order to support our conjecture that a large amount of disk space is wasted in the focused environments, we performed a survey of over 500 instructional machines in our organization. The survey showed that more than 80% of machines have 1.5 GHz Intel Pentium 4 or better processors, and the total available disk space ranged from 8 GB (for older systems) to 60 GB (for the latest systems). A little over 84% of the machines have a local disk of 40 GB. However, the local disk utilization is only up to 4 GB for holding the operating system and temporary user files. Except for older systems that have less total disk space, on average about 90% of the local disk space on each machine is unused. As disks become cheaper and larger in capacity, this wastage is bound to worsen. On the other hand, the three NFS servers used by these machines have about 75% space being used. The servers have to impose strict quotas in order to avoid being full. Such central servers require regular addition of new disk space to accommodate new users and applications, an obviously expensive and cumbersome procedure. These observations stress the opportunity of, and the need for, utilizing the locally available disk space as an economical way of fulfilling the ever-growing storage demands of

users. In the mean time, running NFS servers on each machine with an unused local disk space is far from practical as discussed earlier.

2.2 Structured peer-to-peer overlay networks

Structured peer-to-peer overlay networks such as CAN[26], Chord[31], Pastry[27], and Tapestry[34] effectively implement scalable and fault-tolerant *distributed hash tables* (DHTs), where each node in the network has a unique node identifier (`nodeId`) and each data item stored in the network has a unique key. The `nodeId`s and keys live in the same namespace, and each key is mapped to a unique node in the network. Thus DHTs allow data to be inserted without knowing where it will be stored and requests for data to be routed without requiring any knowledge of where the corresponding data items are stored.

The key aspects of these structured P2P overlays are self-organization, decentralization, redundancy, and routing efficiency. Self-organization promises to eliminate much of the cost, difficulty, and time required to deploy, configure and maintain large-scale distributed systems. The process of securely integrating a node into an existing system, maintaining its integrity invariants as nodes fail and recover, and scaling the number of nodes over many orders of magnitude is fully automated. The heavy reliance on randomization (from hashing) in the `nodeId` and key generation provides good load balancing, diversity, redundancy and robustness without requiring any global coordination or centralized components, which could compromise scalability. In an overlay with N nodes, messages can be routed with $O(logN)$ overlay hops and each node only maintains $O(logN)$ neighbors.

The functionalities provided by DHTs allow for transparent distribution of files on multiple servers. In the next section, we discuss how this facility is leveraged in the Hoard design. While any of the structured DHTs can be used to implement file distribution in Hoard, we use Pastry as example in this work. In the following, we briefly explain the DHT mapping in Pastry.

Pastry Pastry [27, 9] is a scalable, fault resilient and self-organizing peer-to-peer substrate. Each Pastry node has a unique, uniform, randomly assigned `nodeId` in a circular 128-bit identifier space. Given a message and an associated 128-bit key, Pastry reliably routes the message to the live node whose `nodeId` is numerically closest to the key.

In Pastry, each node maintains a routing table that consists of rows of other nodes' `nodeId`s which share different prefixes with the current node's `nodeId`. In addition, each node also maintains a leaf

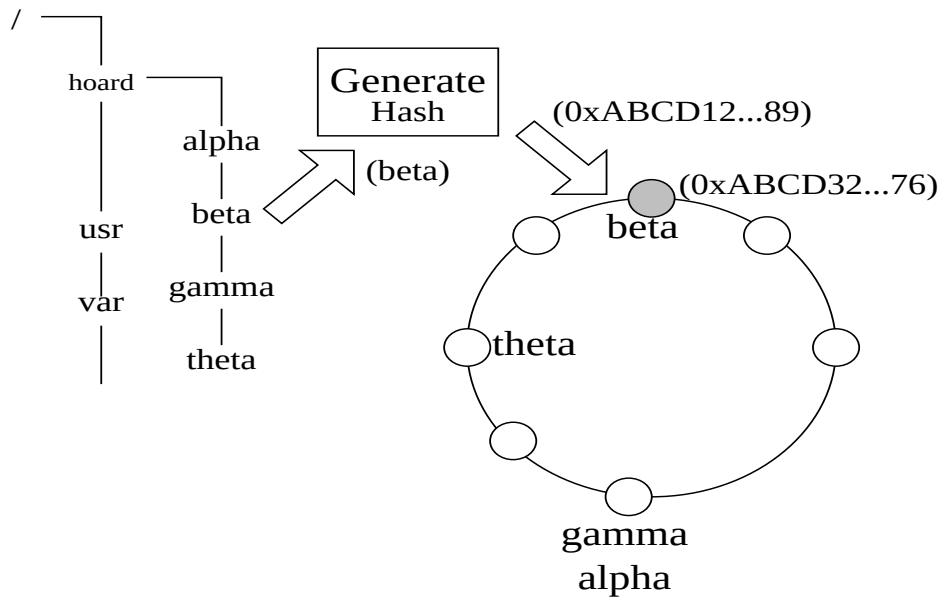

Figure 1. Example of distribution to multiple nodes: The virtual mount point is *~/hoard*. The file/directory name is first hashed using the Generate Hash function to generate a unique key, which is then routed using Pastry to a node whose `nodeId` is the numerically closest to the key. The selected Pastry node will provide the physical storage for the file. The actual file operations, however, are performed via the NFS protocol (not shown).

set, which consists of l nodes with `nodeId`s that are numerically closest to the present node's `nodeId`, with $l/2$ larger and $l/2$ smaller `nodeId`s than the current node's `nodeId`. The leaf set ensures reliable message delivery and is used to store replicas of application objects. Pastry routing is prefix-based. At each routing step, a node seeks to forward the message to a node whose `nodeId` shares with the key a prefix that is at least one digit longer than the current node's shared prefix. The leaf set helps to determine the numerically-closest node once the message has reached the vicinity of that node. A more detailed description of Pastry can be found in [27, 9].

3 Hoard design

Hoard provides the UNIX file system interface to applications/users by leveraging redundant disk space available on various desktops/compute servers, henceforth referred to as nodes. Nodes contributing disk space are assumed to run NFS servers, so that all such nodes can be accessed via NFS. Each

node also runs an instance of the Hoard software. Optionally, the user can also create a local disk partition with a desired size, and use this partition for space contribution. This provides a way for limiting the disk space contributed to Hoard. The contributed space can also be zero, implying that the node will use the storage passively. The nodes constitute a peer-to-peer overlay network, and are identified by unique `nodeId`s assigned to them via the Pastry interface [27]. The nodes are therefore logically arranged on a circle of $2^{128} - 1$ `nodeId`s.

The software creates a virtual mount point (*/hoard*) that serves as an access point to the distributed storage. Hoard distributes the files created under */hoard* to the various nodes contributing unused disk space to the system. The selection of the node on which the file will physically reside is done using hashing and peer-to-peer routing as explained in Section 3.2.1. Figure 1 shows an example distribution of directories to various nodes. Once the node lookup is done, Hoard software creates a NFS mount point and links it to the virtual mount point.

For supporting various operations on a file, Hoard first selects the node on which the file is stored. Standard NFS calls to the selected node are then employed to perform the operation. This guarantees that the whole process remains transparent to the user/application, except for a delay cause by the look-up for the appropriate NFS server.

It can be argued that this process is similar to mounting various files from different NFS servers, however, Hoard provides the following features that serve as main contributions towards creating a truly distributed, reliable file system:

- **NFS semantics**: Hoard exports NFS semantics to the user/application, which makes it possible to provide transparent application access to the distributed storage.

- **Decentralized control**: The nodes in Hoard are arranged in a peer-to-peer configuration, with no single node in charge of the system, and nodes making routing decisions independently.

- **Availability**: Besides choosing appropriate live nodes for storing files, Hoard also employs a replication scheme which replicates a file to K neighbor nodes of the primary node in the identifier space. This enables Hoard to provide near 100% availability even in the event of multiple failures.

- **Load balancing**: Hoard distributes all the subdirectories within an adjustable level from the virtual file system root directory (*/hoard*) among multiple nodes, but storing all the files within each

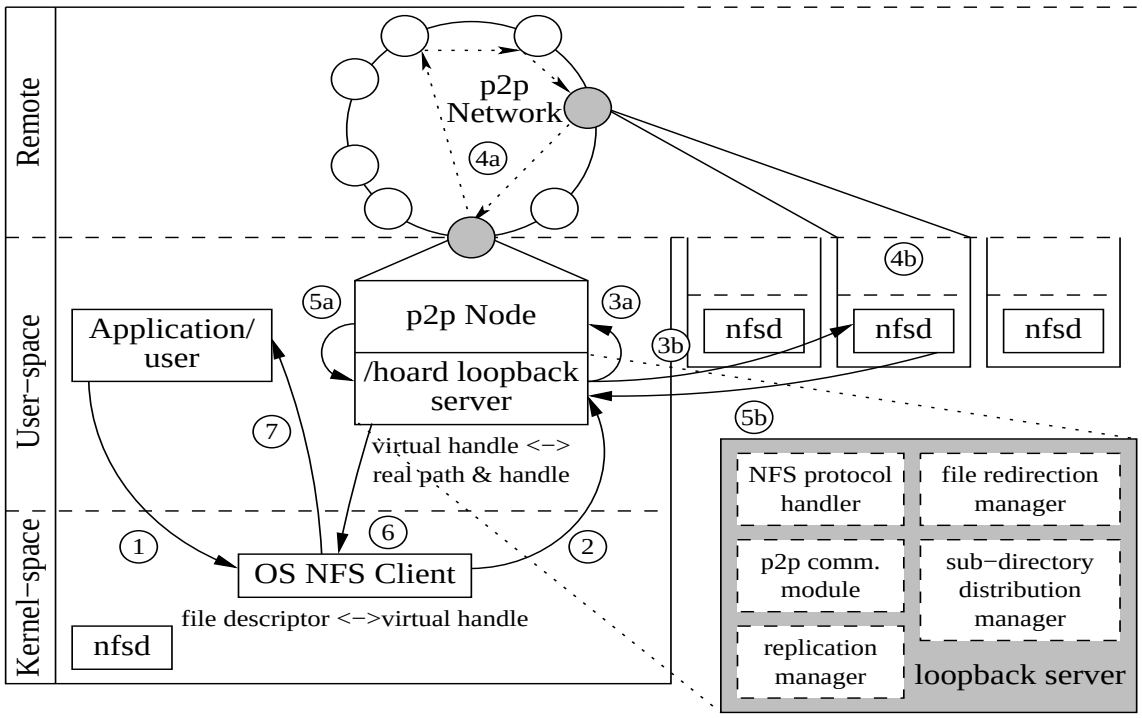

Figure 2. Hoard architecture: (1) application makes an I/O system call, (2) kernel makes an RPC call, (3) (a) local port request to peer substrate or (b) handle substituted and RPC forwarded, (4) (a) overlay locates node storing file or (b) file I/O occurs, (5) (a) local port reply from peer substrate or (b) I/O result returned, (6) RPC returns with virtual handle or result, (7) system call returns control to application.

directory on the same physical node. Hoard also employs file redirection, in case a node can no longer accommodate a newly added file.

3.1 Architectural overview

The Hoard daemon running on each node is implemented as an NFS loopback server [23] tightly-coupled with one of the peer-to-peer routing substrates mentioned in Section 2.2, Pastry, as shown in Figure 2. In the rest of the report, we will often refer to the two parts as one entity, as both are required to run on each participating node.

The loopback server has several components to handle various operations of Hoard. The *NFS protocol handler* manages the RPC to and from the local NFS client as well as remote NFS servers. The *peer-to-peer communication module* interfaces with the local peer-to-peer component of Hoard and is

responsible for all communication between the two components of Hoard. The *subdirectory distribution manager* performs the necessary operations for distributing subdirectories to multiple nodes. It also handles the complex process of renaming distributed directories. The *file-redirection manager* is the component responsible for redirecting files if a node can no longer accommodate it. It handles special link creation as well. Finally, the *replication manager* maintains K replicas of the files, and handles all necessary operations required for this task.

To aggregate unused disk space on participating machines, the Hoard loopback daemon *hoardd* on each machine is assigned to the same virtual mount point, */hoard*. Afterwards, whenever an application performs a file I/O on any path beginning with */hoard* (step 1), the NFS portion of the OS kernel will make a remote procedure call (RPC) to the loopback server *hoardd* (step 2).

On each participating node, */hoard_store* is the data storage directory. It is a sticky directory similar to */tmp*, so that any user may create a file within it, but only the owner of such a file can remove it afterwards. The amount of storage space allocated to */hoard_store* is a per-node configuration parameter. From a user's perspective, the */hoard/$USER* directory actually corresponds to the union of the */hoard_store/$USER* directories on all nodes, as shown in Figure 3.

The `automounter` [8] is used so that the distributed directories on a node can be mounted on demand. Upon accessing a file, the unique node where the file is physically stored is determined. Once this node is known, the `automounter` mounts the aforementioned directories from the selected node under the data storage directory */hoard_store*. A timer is then started. In case the file is not modified for a long time, the storage directory will be unmounted, only to be mounted again on the next access. The system attempts to keep the files in one directory on a single node as long as the node has enough space. This attempt by the system, along with the observation that a typical application usually accesses files in the same working directory, implies that the number of file systems mounted by a node does not become overwhelming.

Finally, the motivation for organizing the machines into a structured peer-to-peer overlay network is to tolerate node failures. This is achieved via three steps. First, the DHT routing is used to map a file name under */hoard* to the local disk of a specific machine. This happens to file operations that operate on file names, such as `lookup`, `create`, and `mkdir`. Second, after a new file or directory is created, the loopback server on the machine of actual creation also creates replicas on neighboring nodes in the

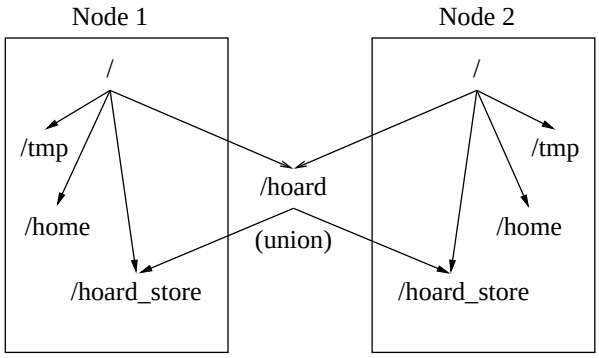

Figure 3. Virtual directory hierarchy: */hoard* **is the virtual directory and is the union of** */hoard_store* **on all the nodes.**

peer-to-peer network. Third, the loopback daemon *hoardd* introduces a level of indirection to all the file access operations by creating *virtual file handles* for files under */hoard* and maintaining the mapping of virtual file handles to real file paths and handles. The local NFS client communicates with *hoardd* only through these virtual file handles. This is feasible since NFS handles are opaque, and thus to the kernel, the virtual handles are not any different from standard NFS file handles. *The extra level of indirection enabled by the use of virtual handles allows Hoard to transparently substitute handles for file replicas in the event of node failure.*

3.2 File and directory distribution

We now describe how files and directories under the Hoard virtual file system are distributed among participating nodes.

3.2.1 Mapping files to nodes

Hoard uses dynamic hashing to determine the node on which a newly created file/directory should be stored. This is achieved as follows. A 128 bit unique key is created via a SHA-1 [1] hash of the file or directory name. Next, this key is used to look-up a node according to the DHT implementation of the peer-to-peer substrate. For example, in the case of Pastry [27], the selected node is the one whose identifier is numerically closest to the key value. The highly unlikely, but possible, event of key collisions only imply that the colliding files/directories will be stored on the same node, and does not pose a problem, as they still have different file/directory names or different paths.

The simple mapping of file/directory names to nodes in this way is suitable for most cases. In some events, however, the selected node may not have enough local disk space to hold the file/directory to be added. When this happens, the file/directory is redirected and stored on a different node. Redirection is done by concatenating a random salt to the file/directory name, and rehashing the new name to find a suitable node. The process repeats till a node with enough disk space is found, or a pre-specified number of retries is exhausted. This approach derives from a similar approach in PAST [28]. If the file/directory is redirected, a special soft link (in the /hoard file system name space) to the redirected file/directory is created in the parent directory. The link serves the purpose of a pointer to the redirected file/directory, and it helps Hoard in listing directory contents (as the name of the link is the same as the name of the redirected files).

A failure may also occur when no individual node has enough space for the file/directory, although the combined free space of all nodes is larger enough. Hoard does not attempt to break a file into smaller blocks (e.g., as in CFS [13]) as that would require another level of indirection in accessing the files and thus make the design much more complex.

3.2.2 Directory distribution across multiple nodes

The heterogeneity in the amount of disk space provided by individual nodes requires that Hoard provide a load balancing mechanism. This is achieved by employing hashing and redirection on individual files when necessary, as discussed in Section 3.2.1. However, if the distribution process is applied to individual files, the cost in terms of hashing and subsequent lookups for the actual storage nodes can become very costly. To reduce such cost while maintaining a good load balance, Hoard assumes that all the files in a directory will be mapped to the same node, the node to which the directory name is mapped, except the subdirectories, which may be mapped using their own subdirectory names and thus may be distributed to different nodes. However, if the current node reaches its contributed space capacity, the new files being created under a directory to the node can be redirected to different nodes. In other words, the distribution of Section 3.2.1 is done at the directory level, unless file redirection is necessitated due to space constraints.

Hoard maintains a system-wide parameter, the distribution level, which dictates how many levels of subdirectories will be distributed to multiple nodes. For instance, distribution level 1 implies that all the

subdirectories under one directory of the virtual file system mount point are stored on the same node, whereas as distribution level 2 implies that another level of subdirectories will also be distributed to multiple nodes.

We use a concrete example to illustrate how Hoard distributes directories instead of files with each directory. Figure 4 shows an example directory hierarchy and the resulting directory distribution. For this discussion we assume that the distribution level is set to 3, and that the nodes have enough disk space so that file redirection is not required.

When the directory */hoard/alpha* is first created, Hoard receives a directory creation request, and examines the complete path. The leading virtual mount point is then removed from the path, in this case leaving */alpha*. The number of '/' in the resulting path is then counted, and indicates the level of the directory, 1 in this case. Since, 1 is less than the distribution level, a node X is located using the directory name only, i.e. *alpha*, and the directory is created on that node. Since this is the first-level directory, no special soft link needs to be created. The directory */hoard/beta* is created similarly. All the files under *alpha* and *beta* are stored on the same nodes to which the parent directories are mapped, respectively.

When */hoard/alpha/sdir1* is created, *sdir1* will be redistributed since it has a directory-level of 2. Hoard finds a node Y by hashing *sdir1* using the DHT. It then creates an empty directory structure, i.e., *alpha*, on node Y, followed by the creation of *alpha/sdir1*. A special soft link *sdir1* is then placed in */hoard/alpha* on node X, and it serves the same purpose as the indirection link in the case of file redirection as discussed in Section 3.2.1, that is, to locate the subdirectories and assist in directory listing.

The reason for creating the directory structure for subdirectories is to distinguish between subdirectories with the same name, for example */alpha/sdir1* and */beta/sdir1*. An additional benefit is to provide access to subdirectories even if their parent directories are inaccessible.

If the distribution level is 2, when */alpha/sdir2/sdirM* is created, no redistribution of *sdirM* will be done, and it will be created on the same node as its parent directory. We look at the effect of varying the distribution level on the overall load distribution in Section 4.3.

In summary, to simplify the directory maintenance, Hoard distributes subdirectories, as opposed to files, within each directory. To avoid unnecessarily fine-grained distribution, it controls the number of levels from from the root file name space beyond which subdirectories will not be redistributed.

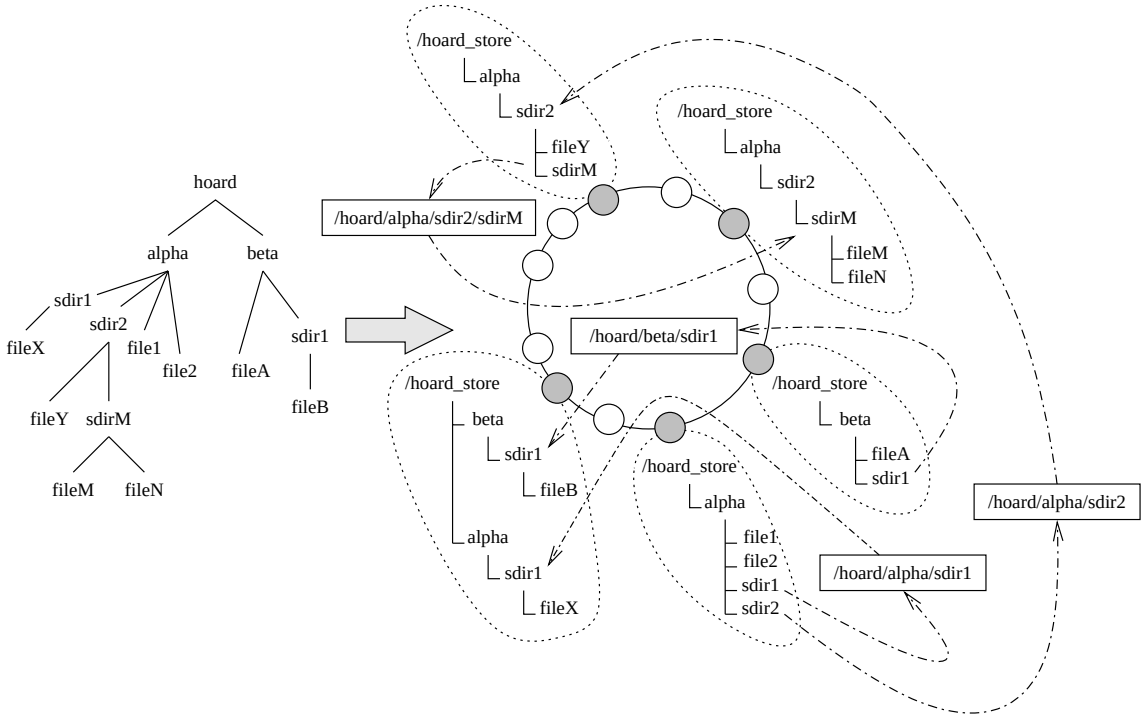

Figure 4. Example of subdirectory distribution to multiple nodes: The files in the same directory are stored on the same node as the parent directory. However, the subdirectories are distributed to remote nodes. A soft link is stored in the parent directory to locate the distributed subdirectory. The link points to a virtual location, which Hoard use to locate the physical location. The example contents of the links are shown in the rectangles. An empty directory hierarchy up to the actual subdirectory is replicated on the node where the subdirectory is stored.

Moreover, a subdirectory can be located using a single lookup with its name, avoiding the cost of an otherwise incremental lookup of the whole path.

3.2.3 Managing replicas

Hoard maintains K replicas of a file on the neighboring K nodes in the node-identifier space. The random assignment of node identifiers ensures that the replicas are fairly dispersed and can provide good fault-tolerance. Hoard employs lazy replication and periodically updates the replica to maintain a relaxed consistency. For each file, there is a primary node, which is in fact the node chosen using the peer-to-peer substrate. Periodically, a node looks at the files/directories it has in its contributed storage. Subdirecto-

ries are not traversed because even if the subdirectories are distributed, the supporting hierarchy is also present, and the replication should maintain the hierarchy by replicating the empty hierarchy along with the subdirectory. For each file/directory, the system calculates the hash and determines from the leaf-set information whether this hash is closer to the node identifier or to its neighbors. The leaf set is assumed to contain at least K neighbors on each side, where K is the number of replicas to be maintained. The difference of the node identifiers and the file hash is calculated and the minimum determined. If this node is the primary node of the file, its updates, if necessary, are pushed to the K neighboring nodes. If the file belongs to a node farther than K, or the primary node of the file has deleted it earlier, the file on this node is also deleted. This deletion is also required for purging of files. For instance, when a node fails, a new replica on a $K + 1$th node will be created, however, when the node is operational again, this $K + 1$th replica is redundant and should be purged. Also, if the node is revived with a new identifier, this scheme ensures that it purges all the old files of which it does not need to maintain replica of. In this way, K replicas of a file are always maintained. If the system cannot sustain K copies of a new file, the newly added files may be added with less than K replicas. The new file creation can be programmed to fail on a minimum replica setting.

The use of lazy replication and file deletion has a further advantage that accidental deletes of important files can also be recovered. For this a version of undelete command is also provided, which retrieves a file from one of the k neighbors to the main node. Similarly, many files have short lifespans [2], and are deleted soon after their creation. The lazy-replication mechanism ensures that such files do not get replicated, and hence load the system.

It should be observed that NFS has strict consistency semantics, and standard file locks can be utilized to enforce single-writer policies as the application desires. However, in the case of failure, the files will revert back to the last periodic replication.

The actual replication is done in the following manner. It is assumed that a distributed time synchronization mechanism such as a Lamport clock [22] is available between all the nodes contributing to the system. In order to do the replication, the main node sends the modification time stamp of a file to the replicas. Each replica compares the timestamp with the version it has, and if it has an older version, it requests an update of the file. Although incremental updates can also be sent, the current implementation copies the entire file when an update is available. Since all the replicas have the same version, a

multicast can be leveraged to update the file on all the replicas in a single data transfer.

3.2.4 Node addition

As nodes join the system, Hoard must dynamically adjust the file distribution to maintain proper location of files. The peer-to-peer component of the system informs Hoard of any node joins/failures in the leaf-set. For this discussion, assume that such notification is received on a node *N*.

Since Hoard leverages DHTs, the nodes are arranged in a ring, with key space between adjacent nodes divided evenly between them. The keys of the files that fall in this key space belongs to one of the neighbors depending on numerical closeness of the file key to the `nodeId`s. No action is required at *N* if the newly added node does not become one of *N*'s immediate neighbors. If a neighbor is added, the key space distribution changes, implying that some of *N*'s files – for which *N* was a primary node – now belong to its new neighbor, and should be moved. For this purpose, Hoard examines the files present on its local storage and uses the leaf-set information to determine which, if any, of the files need to be moved to the new node. If a move is required, the files are instead copied to the new node, and their copy on *N* becomes one of the replicas. The migration of files ensures that a new node always has the files for which it is a primary node. Note that only two nodes, i.e. the immediate neighbors of a newly added node in the key space, will have to act in order to maintain proper operation.

3.2.5 Node failure and fault tolerance

As described in Section 3.2.3, Hoard maintains K replicas of a file which, due to the random nature of the identifiers, may be distributed to diverse nodes. For instance, in an academic environment, the replicas may go to machines in different classrooms or labs, hence providing resiliency to node failures.

The failure scenario occurs when a node goes offline or becomes unreachable (we assume only crash failures), and another node is actively using a file handle mapped to the failed node. With standard NFS, the user is forced to wait for the failed node to be fixed. With Hoard, the NFS client only has a virtual handle and errors are not immediately propagated back to the NFS client. Using a virtual handle allows the real handle to be changed transparently. If *hoardd* detects an RPC error, it removes the mapping for the virtual handle. Alternatively, if one of the nodes within the leaf-set range fail, the Pastry component of Hoard informs *hoardd* about it. In this case, *hoardd* substitutes any virtual handle corresponding to

the failed node with an invalid handle. In any event, when *hoardd* encounters an invalid virtual handle it performs as though a `lookup` RPC was made and locates the handle for another replica of the file. This new handle lookup is automatic, because if a node fails then messages previously routed to it will now go to one of its neighbors. Since, the neighbor already contains a replica; the future access will be to this replica. However, this copy can be from the last periodic replica update, and hence may result in loss of data. Only if no other replicas can be found does *hoardd* return an error to the kernel. By effective replication Hoard provides very high availability, and due to highly randomized location of the neighbors, the probability of finding a replica even under a large number of failures is high. Since the replicas are regenerated, Hoard can tolerate a large number of failures.

There is no support for parity checking or disk striping, although nothing prevents an individual node from doing this on its own.

Storing a mapping from virtual handles to real handles means *hoardd* is not stateless. NFS servers typically minimize their amount of state so in case they fail there is no information to recover later. However, our mapping is only used to provide a service to the kernel running on the same machine. If a *hoardd* fails, then the assumption is the entire machine has failed, including the kernel, so the data intercepted from the RPCs does not need to be persistent.

3.3 NFS operations support

In this section we describe how Hoard handles NFS operations, and what changes are necessary to provide user-transparent access to the file system.

3.3.1 Locating files

hoardd provides location transparency by replacing actual file/directory handles with virtual handles. As mentioned earlier, NFS file handles are opaque; therefore, the value of the virtual handle matters only to the local Hoard and can be as simple as a table index. When an NFSv3 `lookup` RPC arrives at *hoardd* with a virtual parent directory handle and a file/subdirectory name, it expects to receive back the handle for the file. *hoardd* determines the real location of the file, assigns a virtual handle to it, adds a {virtual handle, real path, real handle} triplet to the file handle table for future use, and finally sends the virtual handle as the return value of the `lookup` call. All subsequent RPCs that supply the virtual handle are

translated to use the real handle for the actual NFS operation (see Figure 2 step 3(b)).

For locating files, the requested virtual path is looked up in the table. The virtual path corresponding to an entry in the table is formed by removing any machine-specific prefix from the real path and then appending the filename. For instance, if virtual handle 3 corresponds to the real path */$MACHINE_NAME/hoard_store/a* and the filename is *b*, then the virtual path for the file is */hoard/a/b*. If the requested path is already in the table, the virtual handle is returned and the call completes.

In case the mapping is not found in the table, the parent directory handle is looked up, which may have been obtained through a recent `lookup` call. We can assume that *hoardd* has a table entry for the parent's handle. If not, one can be obtained by repeating the lookup process incrementally on the parent, grandparent etc., at worst until the top level. Once this entry is obtained, the real path of the parent directory is known. Since all of the files under a directory are stored on the same node, and a special link is available in case of subdirectory distribution or file redirection, *hoardd* creates the real path of the file by concatenating the real path of the parent and the filename. For instance, if the real path of the parent directory is */$MACHINE_NAME/hoard_store/x* and the filename is *y*, the real path for the file is */$MACHINE_NAME/hoard_store/x/y*. All subdirectories that are not distributed are treated similarly as files. Using the real path for the file, *hoardd* obtains the real NFS file handle directly from the remote node's *nfsd* (the standard NFS daemon). *hoardd* then examines the returned file handle and determines if it points to a special link – a soft link with special information. If this is not the case, a virtual handle is created for the file, a corresponding entry made in the file handle table, and the RPC completes with the virtual handle as the return value.

If the requested path points to a special link, implying a redirected file or distributed subdirectory, or a top-level directory, peer-to-peer lookup of the remote node is required. In case of a special link, the file/directory name pointed to by the link is used for hashing, whereas for top-level directories their names are used. The local node then uses peer-to-peer routing to send a message to a remote node R, where the file is physically stored as described in Section 3.2.1. This message includes the virtual path as well as some bookkeeping information. R receives the message and replaces the */hoard* part of the path with */$MACHINE_NAME/hoard_store* and obtains a real NFS file handle from its *nfsd* (the standard NFS daemon). R then replies to the local node with a message that contains both the virtual handle and the actual NFS handle. When the reply is received at the local node, the information is extracted and

once again an entry is made in the table and the RPC completes as before.

3.3.2 Listing directories

The `readdir` RPCs provide a virtual directory handle and expect a list of files in the directory. Files/directories under each directory under the */hoard* shared file system (called *virtual directory*) will sometimes be stored on many different nodes due to subdirectory distribution and/or file redirection. Each of the actual directories that constitute a virtual directory is not aware of their status as part of a virtual directory, so special links must be maintained within the virtual directory. These special links have the same name as the file/directory that the user/application created. Hence, the contents of the directory in which they are created are the same as if the files/directories were physically present in the virtual directory. The links are managed exactly as user-stored files, although their permissions are such that only *hoardd* can modify them. Any user access to these links results in the access being performed on the files/directories they point to. The contents of the directory along with the special links are used to compose return values for the RPCs. As a result, for example, doing a `ls /hoard` provides a listing of all the different first-level files/directories present on the distributed nodes – as if all of the files were physically present under the *virtual directory*.

3.3.3 File creation, renaming, and symbolic links

A `create` or `mkdir` RPC arrives at *hoardd* with a virtual directory handle and a filename, expecting the handle for the newly created file or directory. In most cases, the hash of the directory where the file must be created is obtained and a suitable node for the file is selected as discussed in Section 3.2.1. A message is then sent to the selected node. The message carries the virtual path as well as the parameters of the RPC. The receiving node determines the real path, uses the RPC parameters to create the file, and serves as the primary copy of the file. Atomicity of directory operations is preserved since independent actions on the same file will map to the same node. The file handle and path of the created file are sent back to the originator of the message, where they are used as the return value for the RPC.

Since one of the design aspects of Hoard is to only distribute subdirectories, a `rename` RPC of a file does not imply migration to a different node. If the file is not redirected, no special action is required and the `rename` completes in the standard way. If the file is redirected, only the name of the special

link has to be changed, the redirected file itself can continue to be on the remote node it currently is on. It should be noted that this is the main reason for hashing just the parent directory name and not the full virtual path, since it would be inefficient to move files around on every `rename` call. The same process is used for renaming subdirectories that are not distributed.

Renaming of distributed subdirectories is complex, and poses a major challenge. Hoard handles the process of renaming a distributed directory OLD_D to NEW_D in the following manner. First, a node is located using NEW_D. The empty directory structure is then created on the node, followed by the copying of the contents of OLD_D to NEW_D. The special link in the parent directory is then updated to point to NEW_D. OLD_D is then deleted, and replaced by a temporary link to NEW_D. This link is required because any subdirectories of NEW_D are stored with the previous empty directory structure, and does not yet have the correct back pointers to NEW_D. The subdirectories of NEW_D are then traversed, and the empty directory structures associated with them are updated in the similar manner. Since in case of subdirectories the names are not changed, all that is needed is to rename the occurrence of OLD_D in the paths to NEW_D. Even so, the process can be expensive as traversal of all subdirectory levels is required and has to be done on any maintained replicas as well. Finally, the temporary link from OLD_D to NEW_D is removed. It should be noted that this directory renaming is equivalent to a copy to a new location followed by a delete of the old location. This is in compliance with how the rename command works in NFS where a rename is in fact a move which is implemented via a copy to the new destination, followed by a delete of the original source (see `man mv`).

The `symlink` and `readlink` RPCs are supported, although `link` is not since hard links make no sense in our file system.

3.3.4 Reading/writing files and attributes

The `read` and `write` RPCs arrive with a virtual file handle, an offset, a number of bytes, and (for `write`) data. Assuming no failures, the table contains a valid real handle to substitute for the virtual handle, and the real path to determine the machine storing the file. The substitution is made and the call is forwarded to the node storing the file. RPCs `getattr` and `setattr` work similarly. Replicas of the file are updated according to Section 3.2.3. What happens in the event of a failure is discussed in Section 3.2.5.

3.3.5 Removing files

A `remove` or `rmdir` RPC arrives with a virtual directory handle and a filename, expecting a success/error value. The file is deleted from the main node, and the K nodes on which the replicas are maintained are informed of the deletion. However, the nodes do not delete the file immediately as discussed in the next section; although they will pretend as if they no longer have the file. Once the file is removed from the primary node, any access to it returns a `file does not exist` error. This is true, even if the replicas have not yet deleted the file. In case a new file with the same name is created, this new file will be pushed to the replicas. The replicas will treat such changes as updates.

The subdirectories that are not distributed are deleted in a similar manner as files. In case of distributed directories, subdirectories are also traversed and removed. The empty directory structure created to support the distributed subdirectory is then examined for a possible use by other subdirectories with some common path prefix. The entire empty hierarchy leading to the subdirectory is then deleted, and the special link in the parent directory is removed. The K replicas of the directories are also informed, as is the case for file deletion. This completes the directory deletion process.

3.3.6 Semantics

Assuming no failures, every user sees the same instance of a file since all messages concerning the file will route to the same node. Hoard's semantics are the same as NFS when there are no failures. With NFS, files are inaccessible under failure, but Hoard can still provide access. The behavior of Hoard in the presence of client caching also remains the same as that of NFS.

3.3.7 Security

Security in Hoard is identical to NFS since files in Hoard maintain their permissions. The replicas are inaccessible to the local users, as they may accidentally or maliciously modify the replica and hence cause it to be never updated. Also, in most of the targeted academic or cooperate networks, the users are not given administrative access to their machines, or central NFS is not supplied by such machines. Therefore, it is safe to assume that the files stored on distributed nodes are at least as secure as the central NFS server. For added security, however, Hoard can be extended to support a majority consensus system based on Byzantine agreements [10], as utilized in [29]. The performance of the system may

Benchmark	NFS	Hoard							
		One Node		Two Nodes		Four Nodes		Eight Nodes	
		exec. time	ovrhd	exec. time	ovrhd	exec. time	ovrhd	exec. time	ovrhd
mkdir	0.238	0.243	1.02	0.266	1.12	0.287	1.21	0.327	1.37
copy	5.628	5.666	1.01	5.698	1.01	5.719	1.02	5.972	1.06
stat	0.401	1.221	3.05	1.380	3.44	1.384	3.45	1.427	3.56
grep	2.646	2.837	1.07	2.929	1.11	3.146	1.19	3.758	1.42
compile	18.699	22.400	1.20	22.409	1.20	22.410	1.20	23.071	1.23
Total	27.612	32.368	1.17	32.682	1.18	32.945	1.19	34.554	1.25

Table 1. Performance of a modified Andrew benchmark on Hoard with increasing number of nodes. The table shows execution times of each phase in seconds and respective overhead compared to NFS. The distribution level for Hoard was fixed at 1 **for these measurements.**

be sacrificed, if the need for supporting mutually untrusted nodes arises. The peer-to-peer substrate can support such extensions; however, in our implementation we did not incorporate such an approach.

4 Evaluation

In this section we present experimental results obtained from a prototype implementation of Hoard. We first compare the performance of Hoard to that of NFS using a modified Andrew benchmark [29]. We then study the load-balancing capability of Hoard using a file system trace collected from our organization's NFS servers. Finally, we measure the availability of files stored in Hoard using the availability trace of machines in a large organization.

4.1 Prototype implementation

The implementation of Hoard is divided into two parts. One part is dedicated to managing peer-to-peer communication between nodes and utilizes the Pastry API. We used FreePastry [15] in our prototype implementation. Since the FreePastry is written in Java, this component of Hoard is also written in Java (as an application built on top of Pastry) and accounts for $1000+$ lines of code. The second and largest part of Hoard handles accesses to the file system and manages NFS remote procedure calls (RPCs). It is implemented as an NFS loopback server built on top of the SFS toolkit [23]. The event driven asynchronous I/O services provided by `libasync` [24] are utilized to create an efficient

and robust server. For reasons of performance and compatibility with the SFS toolkit, this component is implemented in C++ and accounts for $4000+$ lines of code. This part is referred to as *hoardd*.

In order to start the system, the peer-to-peer part is started first, followed by the execution of *hoardd*. Once started, *hoardd* establishes communication with the local peer-to-peer component using sockets. The messaging between the nodes occurs at two levels. The node lookup and other peer-to-peer messages are relayed using the peer-to-peer substrate. However, once a node is chosen for a specific operation, *hoardd* uses direct NFS RPCs to communicate to remote NFS servers.

The communication is normally initiated by *hoardd* and the peer-to-peer component replies. One exception to this is when a node joins or fails, in which case, the peer-to-peer component informs *hoardd* of the node join or failure. *hoardd* then performs the actions as described in Sections 3.2.4 and 3.2.5.

4.2 Performance

To determine the performance of the proposed scheme, we measured Hoard execution times for a modified [1] Andrew benchmark [29] and compared it to plain NFS. These experiments were performed on a 8-node configuration. Each node is a 2.0 GHz Intel P4 with 512 MB RAM and a 40 GB 7200 RPM Barracuda Seagate hard disk, and runs FreeBSD 4.6.

Table 1 shows the first set of measurements comparing the performance of Hoard, varying the number of nodes, relative to that of NFS. In this case, the distribution level was fixed at 1, i.e., only the first level directories under */hoard* were distributed to multiple nodes. This was done to remove the effect of subdirectory distribution, and thus isolate the performance overhead due to peer-to-peer lookups. Moreover, each node contributed 35 GB of disk space, enough to accommodate all the files to be stored on it, hence eliminating the effect of file redirection. For each Hoard size, 50 runs of the benchmark were made, and the execution time for each phase was recorded. For Hoard, we measured the performance as we successively added nodes 1 to 8. The NFS configuration consists of two nodes with one running as a client, and the other running as a server.

The total overhead introduced by Hoard, as compared to the performance of NFS, is under 25%. The `stat` phase is most affected, while the `copy` phase is least affected. Adding more nodes into the system does not affect the overall performance drastically (only 2% additional total overhead introduced

[1] The benchmark was modified to run on FreeBSD.

Benchmark	Dist-level 1	Dist-level 2		Dist-level 3		Dist-level 4	
	exec. time	exec. time	overhead	exec. time	overhead	exec. time	overhead
mkdir	0.286	0.329	1.15	0.337	1.17	0.343	1.19
copy	5.719	6.466	1.13	7.046	1.23	7.061	1.23
stat	1.384	1.730	1.25	1.790	1.29	1.791	1.29
grep	3.146	3.224	1.03	3.226	1.03	3.263	1.04
compile	22.410	22.860	1.02	23.590	1.05	23.893	1.07
Total	32.945	34.610	1.05	35.989	1.09	36.352	1.10

Table 2. Performance of a modified Andrew benchmark on Hoard as the distribution level is increased. For these measurements, the number of nodes was fixed at 4. All times are in seconds.

on addition of the eighth node), this is because the DHT lookup is always one hop in the small peer-to-peer overlay.

The overhead of Hoard can be explained as follows. For the case of one-node and distribution level 1, the overhead is 4.76 seconds. The modified Andrew benchmark made a total of 39445 different RPCs. As explained in Section 3.3.1, not all of these RPCs required peer-to-peer lookups to determine the actual physical location. At this distribution level, 24 `lookup` RPCs were generated that required the peer-to-peer remote node lookup. The cost of performing this lookup, including the SHA-1 hashing, was an average of 2.3ms, giving a total of 55.2ms. The rest, and main part of the time, in this case, was spent in *hoardd*. The cost of translating the virtual handle to the physical handle, and forwarding the request to the appropriate remote node, on average, is 93.2μs per RPC, which accounts for 3.68 seconds spent in the server. The remaining cost was within the various bookkeeping portions of *hoardd*.

The extra 314ms introduced when the second node was added is mainly due to bookkeeping tasks in *hoardd*. Other factors such as peer-to-peer lookups were negligibly affected. The incremental overhead for the four-node and eight-node cases are also due to added overhead to the `automounter` process.

To measure the effect of subdirectory distribution on the overall performance, we varied the distribution level between 1 to 4, while fixing the number of nodes in Hoard to be 4. Once again, 50 runs of the Andrew benchmark were made, and the execution times were recorded. The reason that the distribution level was not increased beyond 4 is because the benchmark directory hierarchy had only up to 4 levels

of subdirectories, and thus increasing the distribution level would have no effect.

Table 2 shows that the overhead in distribution levels 2, 3, and 4 relative to distribution level 1 are 5%, 9%, and 10%, respectively. This implies that having a large distribution level is not inhibiting. Also observe that the cost on `mkdir` and `copy` is significantly more than on `compile` and `grep`. The reason for this is that when the directories are created in `mkdir` and/or `copy` phases, Hoard has to perform multiple hashing to locate the node on which the subdirectory will be stored, and to locate the parent directory where the special link will be created. Then the empty hierarchy as well as the special link have to be created, adding to the overall cost. On the other hand, during `compile` phase for instance, only one hash of the directory name results in the location of the physical node storing the file. It should also be noted that if individual files were distributed, hashing for each file would have been required, and the cost of performing similar operations on them would be very high.

A breakdown of the additional overhead of 1.67 seconds when the distribution level is increased from 1 to 2 is as follows. The main factor was the increase in special operation related to `mkdir` and `lookup` for handling distributed subdirectories. There were 11 subdirectories that were distributed at this level. The distribution resulted in an additional 11 `symlink`, 11 `lookup`s that also required peer-to-peer routing, and 16 `mkdir` RPCs to generate the empty directory structure. The cost for these operations was 67ms. There were additional 28 `lookup` RPCs for distributed subdirectories which also required peer-to-peer routing and cost 65ms. The rest of the time (1.54 seconds) was spent in *hoardd* for operations such as distribution level determination and bookkeeping of special operations. Increasing the distribution level further increases these peer-to-peer operations and hence the overheads.

It was observed that in the experiments, the RPC calls that do not involve peer-to-peer hashing cost on average 923μs, out of which about 93μs is the overhead introduced by *hoardd*. For other RPC calls which involve peer-to-peer hashing, the Pastry routing introduced a high overhead of 2ms to 3ms per call. This high overhead is due to the Java implementation and use of Remote Method Invocation (RMI) in the Pastry portion of Hoard. For example, a simple measurement of `null` RMI calls between two Hoard nodes yielded an average roundtrip time of 706.42μs, where as the similar `null` RPC only took 368.63μs. Given the high number of RPCs, this difference also becomes significant.

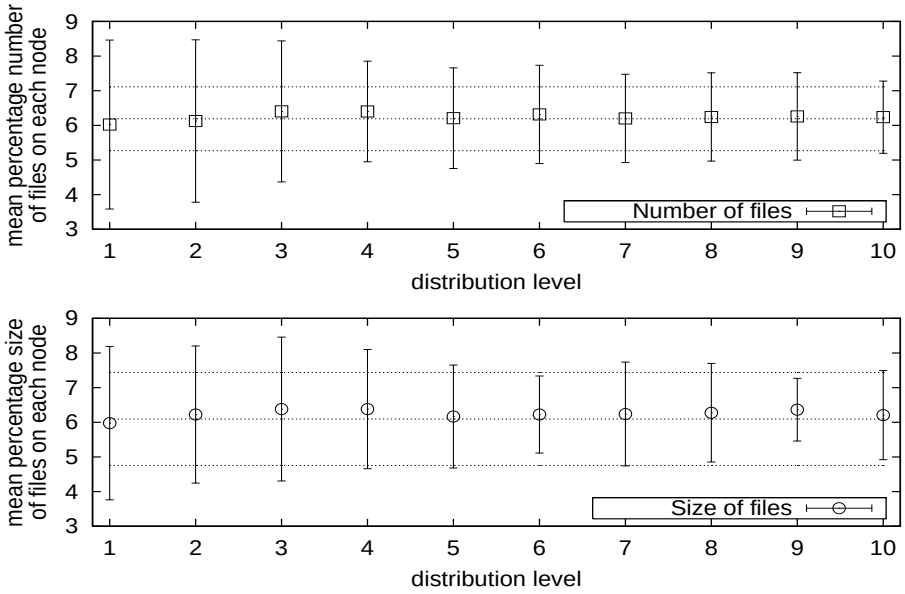

Figure 5. The mean and standard deviation of the percentage of number of files and their sizes across 16 nodes as the distribution level is increased. The dotted horizontal lines show the mean and the standard deviation when each individual file was distributed to a different node, i.e., the finest-grained distribution.

4.3 Load distribution

The load distribution facilities of Hoard are evaluated in this section. For the purpose of these evaluations, we simulated a Hoard cluster of 16 nodes and fixed the number of replicas to 3. The simulation was driven by a file system trace, which we collected from the central NFS server of our organization. The trace contained 221K files of 130 users, for a total of 17.9 GB of data.

The first set of experiments measured the effect of subdirectory distribution on the load-balancing characteristics of the system. Each node contributed 10 GB of disk space to avoid file redirection. The distribution level was varied from 1 to 10, and for each level, we collected the distribution information from all nodes, and measured the number of files and their collective sizes on the individual nodes. The simulation was repeated 50 times varying the nodeId assignments in the Pastry network, and the results were averaged. We also calculated these quantities for a hypothetical scheme which distributed individual files among different nodes. This finest-grained approach serves as an upper bound on the best load balancing (for the trace used) that can be achieved using DHTs.

Figure 5 shows the result of the load-balancing experiments. The dotted horizontal lines show the

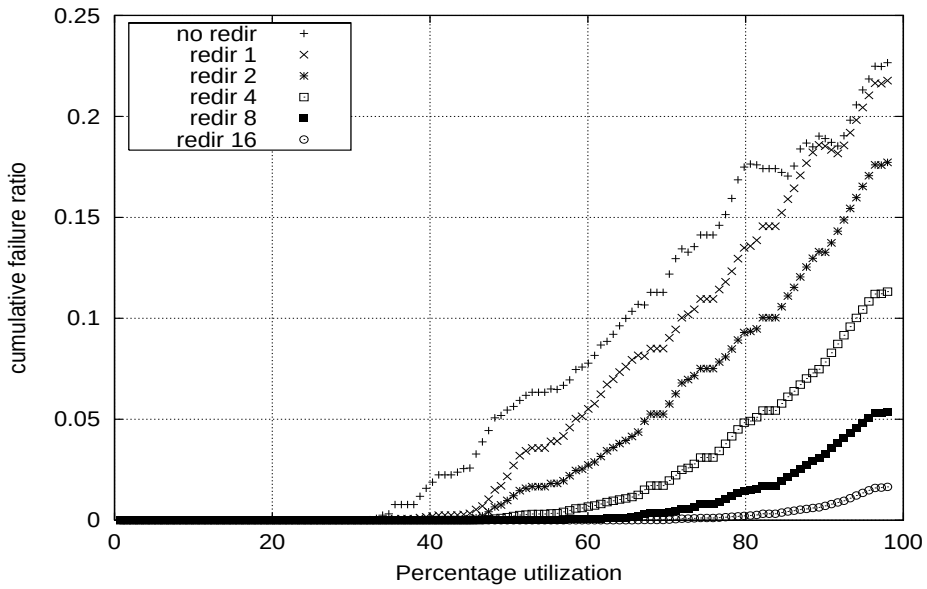

Figure 6. The cumulative failure ratio versus utilization, as the number of redirection attempts is increased. The distribution level is fixed at 4.

mean and the standard deviation of the distributions of the number and the collective sizes on the different nodes when each individual file was hashed and distributed. The results show that as the distribution level is increased, the load balancing in terms of the number of files converges towards the best case. The file size distribution improves, but the improvement is not monotonic. This is because the distribution process is not based on file sizes. Using directory distribution with distribution level 4 or greater provides comparable load balancing to that of individually hashing all files.

The next set of experiments measured the effect of file redirection on the overall disk utilization. The simulation for this was done for a cluster of 16 nodes, 8 of which contributed 3 GB each, 4 nodes contributed 4 GB each, and 4 nodes contributed 5 GB each of disk space. These numbers were chosen to study the system under high utilization. The distribution level was fixed at 4, and the number of the replicas was fixed at 3. The file system trace from our organization was once again used to drive the simulation, and the number of insertion failures was recorded as the files were added. The simulation was repeated with file redirection attempts varying from 1 to 16. Each simulation was run 50 times varying nodeId assignment in the Pastry network, and the results were averaged. In [28], the cumulative failure ratio is defined as the ratio of all failed file insertions over all file insertions that have occurred up to the point when the given storage utilization was reached. We use the same definition. Figure 6 shows the

cumulative failure ratio versus the percentage utilization. It shows that with 4 redirection attempts and distribution level 4, the failure ratio remains near 0 for utilization as high as 60%, and it does exceed 12% when the utilization approaches 100%. Note that while increasing the number of redirection attempts results in a higher utilization of the total disk space, each redirection attempt requires hashing of the file name which can hinder the file operation performance.

4.4 Fault tolerance

The experiments in this Section measure the availability of Hoard under failures. We used an availability trace of 51663 machines in a large corporation over a consecutive 35-day (840 hours) period [5]. The trace contains the status of machines (up or failed) recorded hourly. We simulated Hoard for the cluster of 51663 machines. We distributed the files obtained from the file-system trace from our organization's servers as described earlier, and then used the availability data to introduce failures and node joins. For each hour, we determined the total number of files that remain available. The distribution level was fixed at 3, and the experiments were repeated with number of replicas varying from 0 to 4. For each case, 100 runs were made with various nodeIds for the nodes in the Pastry network, and the results were averaged.

Figure 7 shows the percentage of total files available over the 840 hours period. The lower spike in the graph for Hoard-0, i.e., with no replicas, shows that the system performance is affected when a large number of failures occur. However, even maintaining a single replica (Hoard-1) increases the availability significantly, even for the case of large number of simultaneous failures at hour 615. For the case of Hoard-3, the average availability is 99.991%, signifying that Hoard can guarantee near 100% availability with only three replicas. The reason for this is that Hoard contiguously maintains the K replicas it was configured for (Section 3.2.3); node failures are tolerated as new replicas are created when old ones become unavailable.

5 Related Work

The main driving force behind widespread use of peer-to-peer techniques has been large scale data sharing facilities such as Gnutella [32], Freenet [11], and Kazaa [19]. The basic data sharing is extended by providing strong persistence, and reliability in peer-to-peer distributed storage projects, such

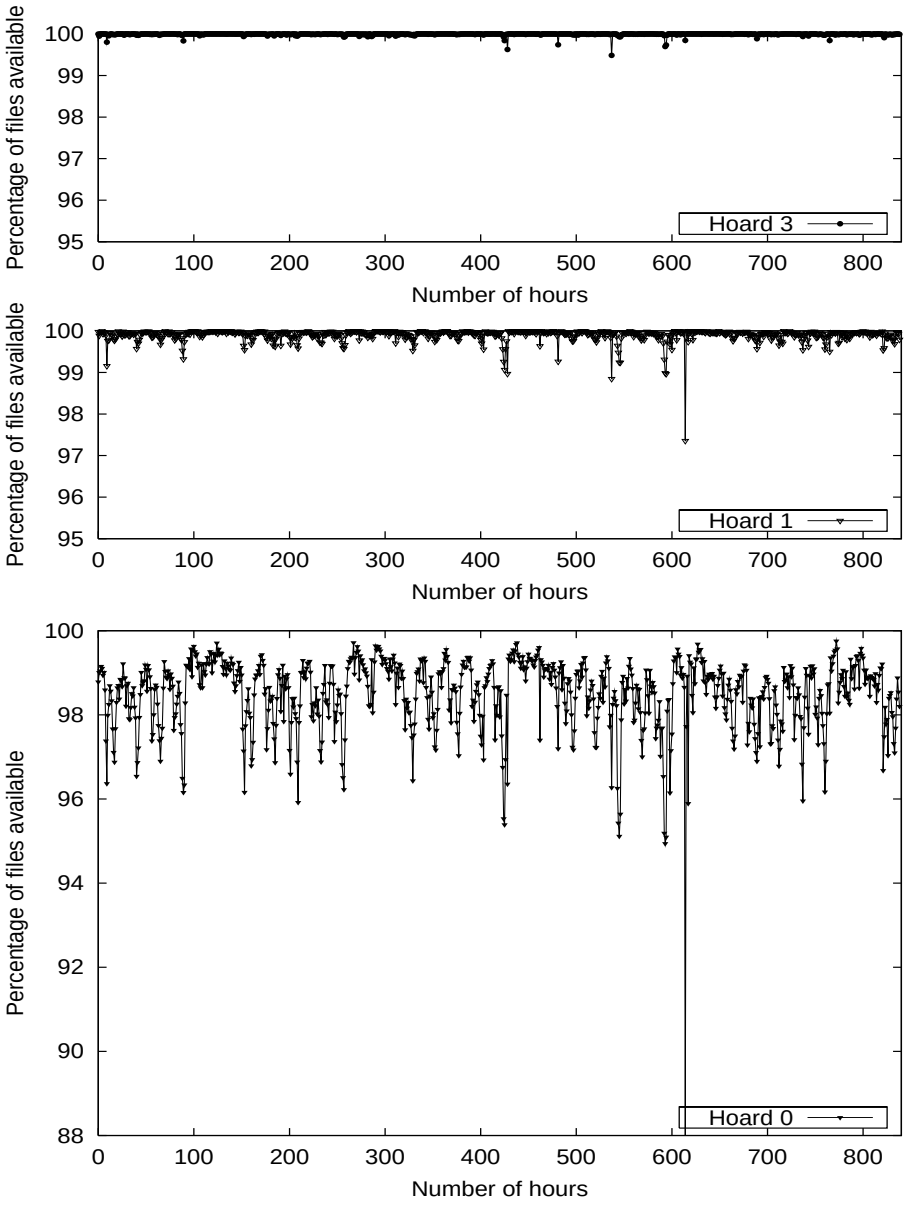

Figure 7. Percentage of total files that are available over a period of 840 hours. The distribution level was fixed at 3 for these results. The largest number of failures (4890) occurred at hour 615, where over 12% files became unavailable for Hoard-0 compared to only 0.16% for Hoard-3.

as Oceanstore [21], CFS [13], and PAST [28]. Hoard uses a similar peer-to-peer substrate but also provides a virtualized NFS interface that creates a file system abstraction to the distributed storage.

Coda [20] provides a reliable file system by replicating files on multiple servers. There are several wide-area file system projects such as Ivy [25], Farsite [2], and Pangaea [29], which also provide reliability. In contrast to these file systems, Hoard does not focus on wide-area scalability. Instead, it focuses on extending the capabilities of a local-area NFS. This approach is more likely to see actual use since wide-area file storage raises more issues of trust and consistency, despite the numerous approaches that have been developed to address these problems, such as encryption [11], agreement protocols [10, 6], and logs [25]. Hoard avoids most of these problems since it is only concerned with the consistency of replicas stored within a common NFS. The local file system permissions obviate encryption and lock files will work normally.

Among the wide-area file systems, the closest to Hoard is Pangaea [29], which also provides the NFS interface to the user via a loopback server. It utilizes randomized graph algorithms to manage large groups of replicas in a wide-area network, and only provides very weak consistency. Hoard on the other hand uses a structured peer-to-peer overlay to maintain replicas, while providing the same consistency as NFS.

Hoard uses the loopback server built on top of the SFS toolkit [23], similar to SFS [16]. However, SFS has a different objective than Hoard; it provides a secure and reliable file system which can withstand malicious servers and/or networks.

Slice [3] is designed to provide high bandwidth shared storage using dedicated network storage nodes. It uses request routing policies enforced via an interposed request switching filter, and presents the user with a transparent NFS interface. Hoard shares with Slice common issues on distributing files and directories among multiple nodes, but differs from it in that files in Hoard are distributed among peering nodes contributing unused storage, while in Slice files are distributed among back-end storage servers.

Scalable distributed [18] or serverless [4, 33] file systems provide some peer-to-peer aspects, but may not be practical to switch to in an established environment because of their fundamentally different designs and requirements. NFS is well-tested and widely used, and we believe augmenting it to include these capabilities will be much more likely to see actual use.

6 Conclusion

We have presented Hoard, a peer-to-peer enhancement for the widely-used network file system (NFS). By blending the strengths of NFS with those of peer-to-peer overlays, Hoard aggregates unused disk space on many computers within an organization into a single, shared file system, while maintaining normal NFS semantics. In addition, Hoard provides location transparency, mobility transparency, load balancing, and high availability through replication and transparent fault handling. We have built a Hoard prototype on top of the SFS toolkit [23], using the Pastry peer-to-peer overlay for node location in distributing directories. Performance measurements in a LAN show that Hoard over eight nodes incurs total overhead of 25%. Simulations using a large file system trace shows that Hoard directory-distribution techniques achieves a balanced load distribution similar to that of distributing individual files. Simulations using a machine availability trace collected in a large organization show that with Hoard guarantees near 100% availability during node failures by maintaining three replicas of each stored file. Since Hoard exports the NFS interface and consistency semantics, it is more likely to see actual use than techniques that provide fundamentally different interfaces.

References

[1] F. 180-1. Secure Hash Standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), NIST, US Department of Commerce, Washington D.C., April 1995.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. OSDI*, December 2002.

[3] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Proc. OSDI*, October 2000.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1), February 1996.

[5] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed system deployed on an existing set of desktop pcs. In *Proc. SIGMETRICS*, 2000.

[6] D. Brodsky, J. Pomkoski, M. Feely, N. Hutchinson, and A. Brodsky. Using versioning to simplify the implementation of a highly-available file system. Technical Report TR-2001-07, 23 2001.

[7] B. Callaghan. *NFS Illustrated*. Addison Wesley Longman, Inc., 2000.

[8] B. Callaghan and T. Lyon. The Automounter. In *Proc. 1989 Winter USENIX Technical Conf.*, Jan 1989.

[9] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical report, Technical report MSR-TR-2002-82, 2002, 2002.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, 1999.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *http://freenetproject.org*.

[12] Compaq. http://www.compaq.com/.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, October 2001.

[14] Dell Computer Corporation. http://www.dell.com/.

[15] FreePastry. http://www.cs.rice.edu/CS/Systems/Pastry/ FreePastry/.

[16] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proc. OSDI*, October 2000.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (3rd Edition)*. Morgan Kaufmann Publishers, 2002.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[19] Kazaa. http://www.kazaa.com/.

[20] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[21] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, D. G. P. Eaton, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS*, 2000.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[23] D. Mazieres. A toolkit for user-level file systems. In *Proc. USENIX Technical Conference*, June 2001.

[24] D. Mazieres, F. Dabek, E. Peterson, and T. M. Gil. Using libasync. http://www.pdos.lcs.mit.edu/6.824/doc/libasync.ps.

[25] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *OSDI*, December 2002.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. SIGCOMM*, August 2001.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, October 2001.

[29] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. OSDI*, December 2002.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proc. Summer USENIX*, pages 119–130, June 1985.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, August 2001.

[32] The Gnutella protocol specification v0.4 2000. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.

[34] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.